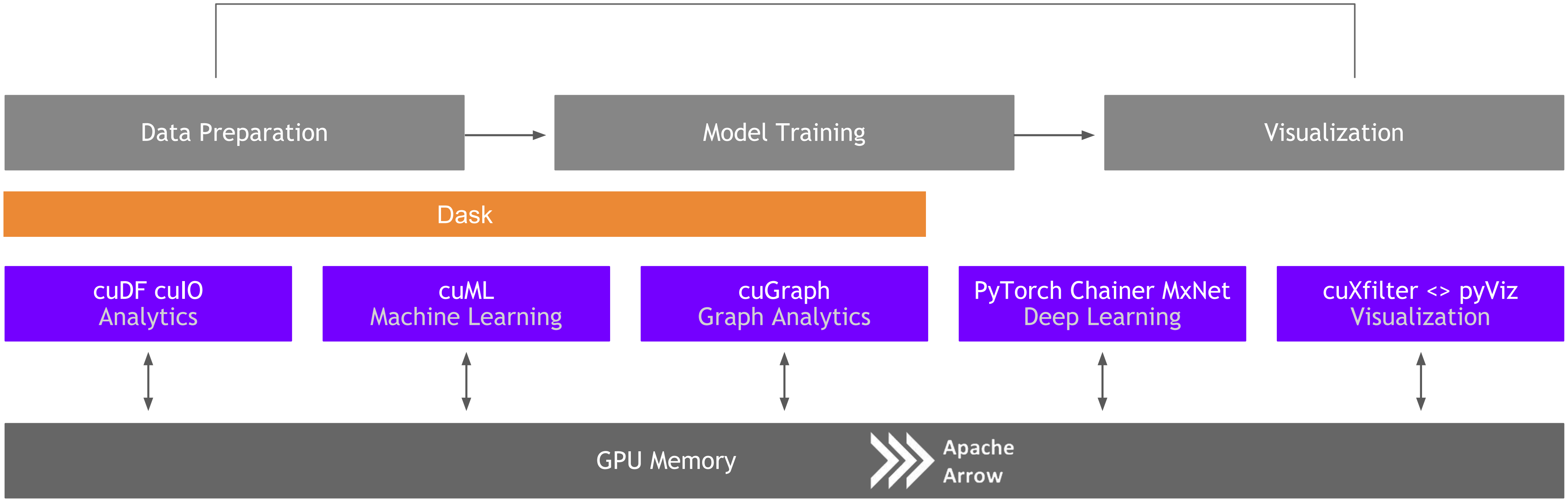
End-to-end data-science and geospatial analytics with GPUs, RAPIDS, and Apache Arrow

Joshua Patterson – GM, Data Science

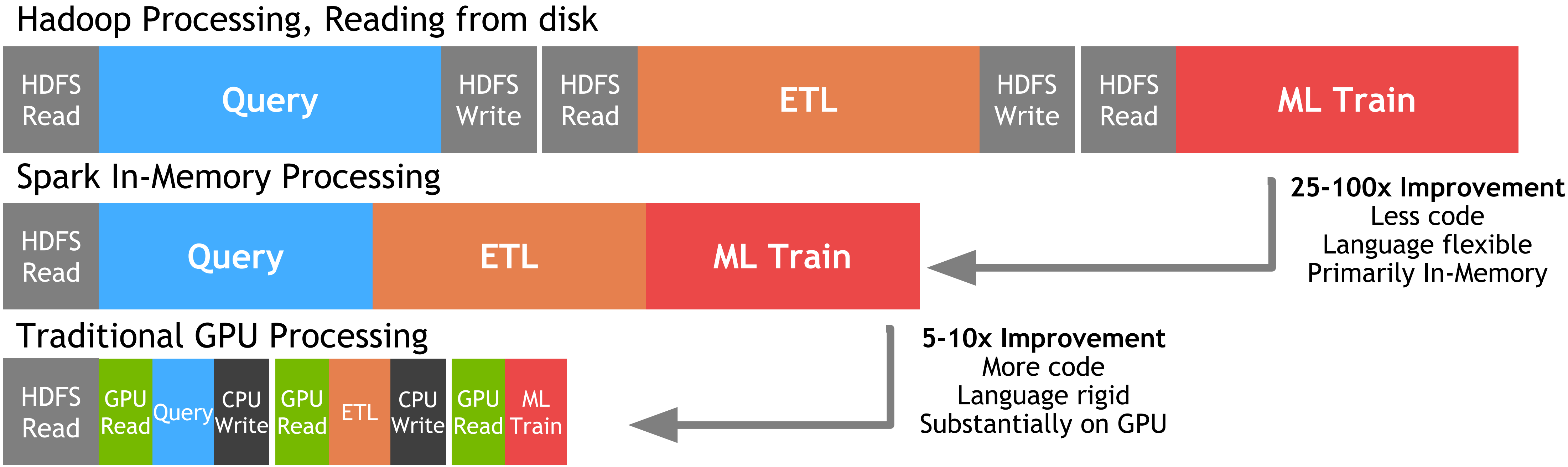
RAPIDS

End-to-End Accelerated GPU Data Science



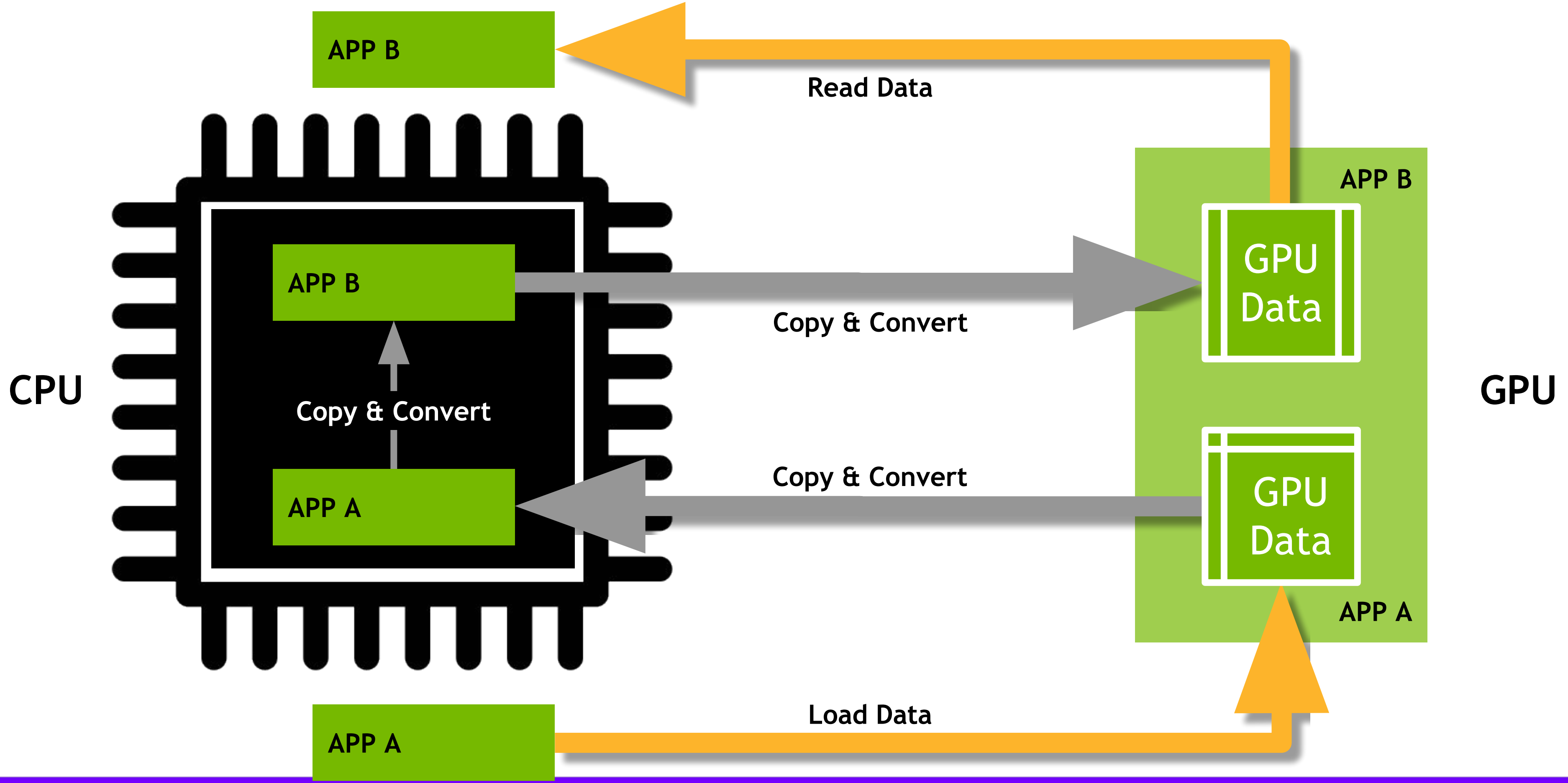
Data Processing Evolution

Faster data access, less data movement



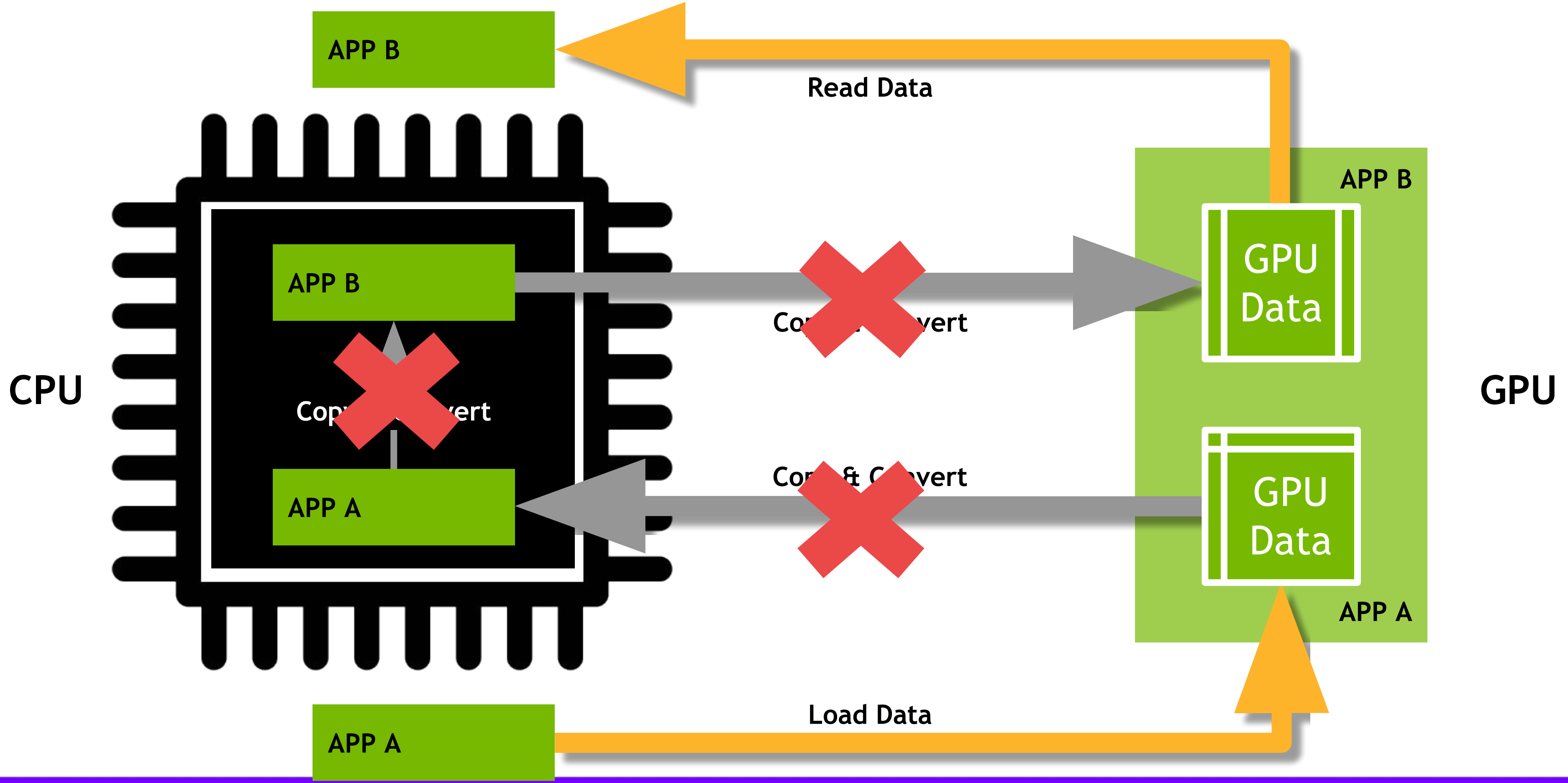
Data Movement and Transformation

The bane of productivity and performance

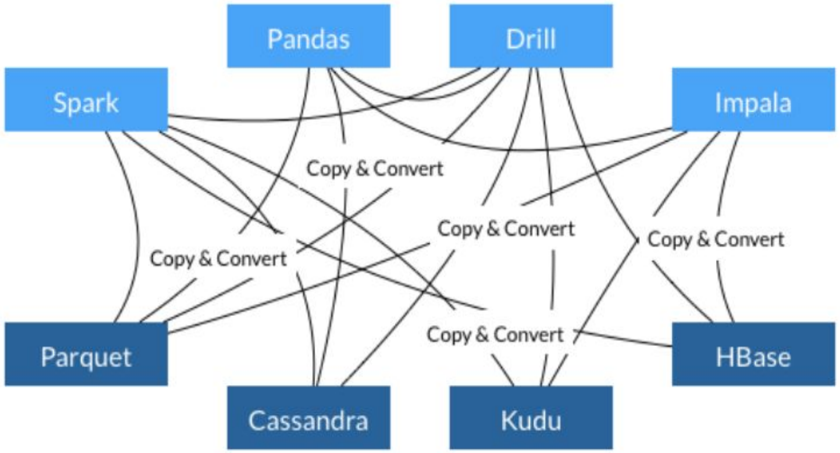


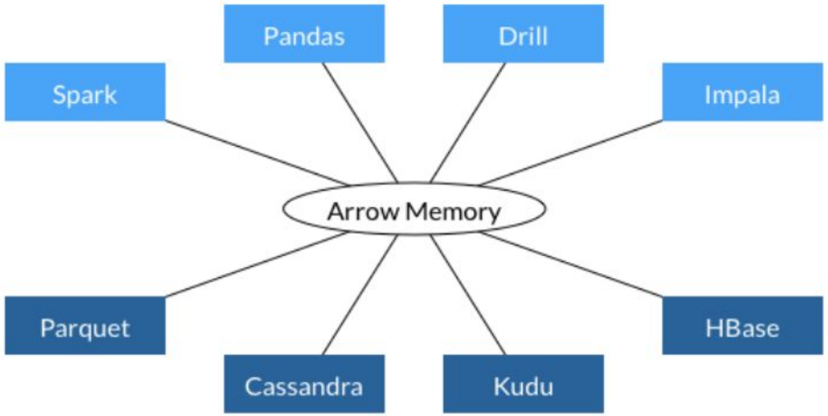
Data Movement and Transformation

What if we could keep data on the GPU?



Learning from Apache Arrow





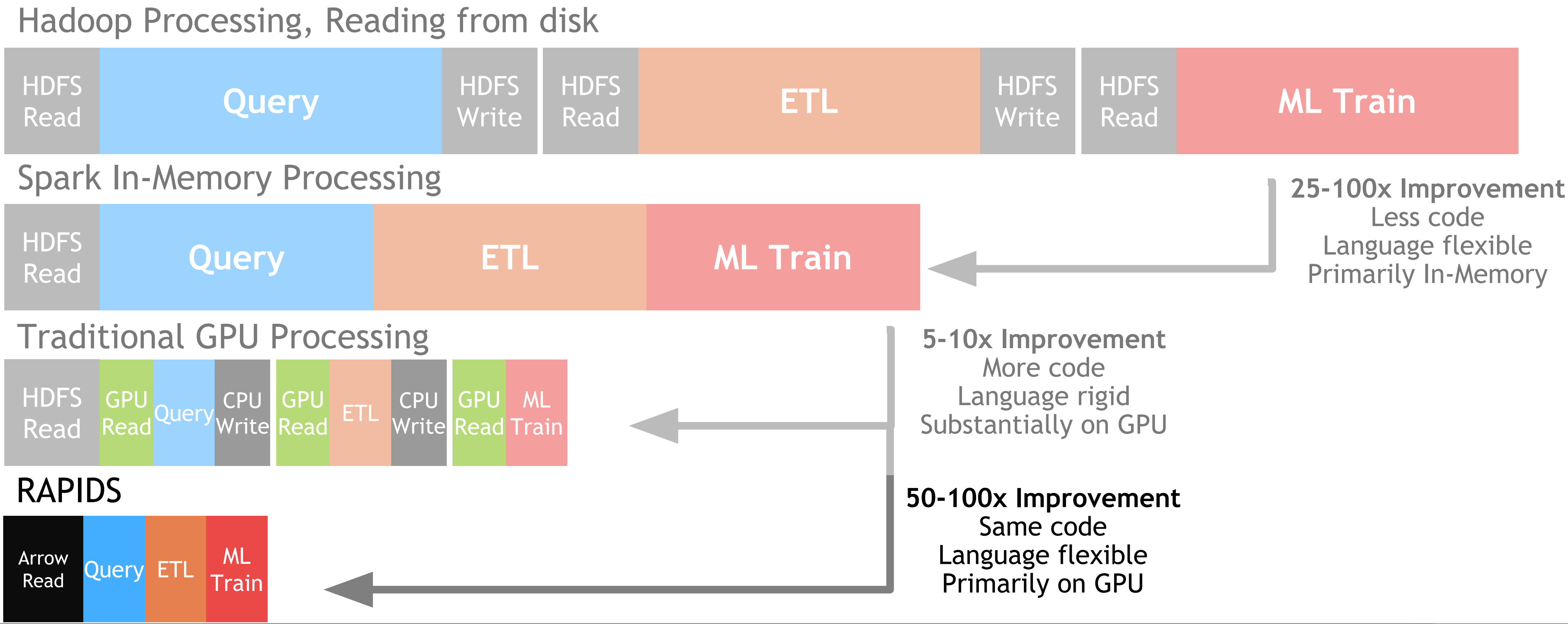
- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg, Parquet-to-Arrow reader)

From Apache Arrow Home Page - https://arrow.apache.org/

Data Processing Evolution

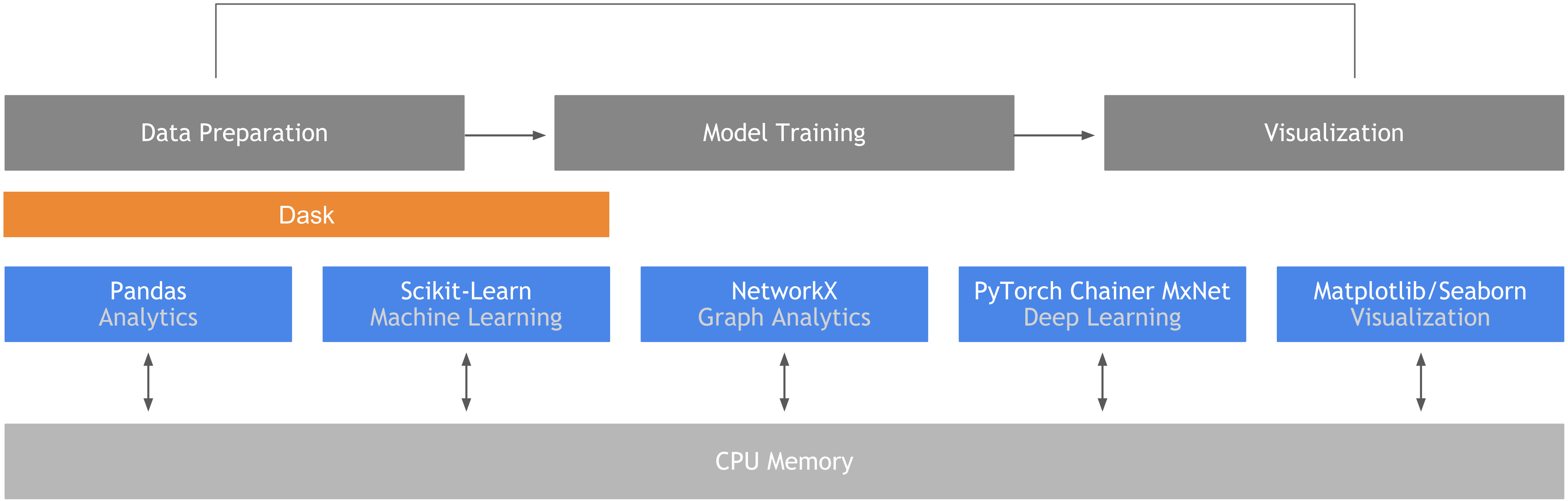
Faster data access, less data movement



RAPIDS Core

Open Source Data Science Ecosystem

Familiar Python APIs



RAPIDS

End-to-End Accelerated GPU Data Science

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO
Analytics

cuML
Machine Learning

cuGraph
Graph Analytics

PyTorch Chainer MxNet
Deep Learning

cuXfilter <> pyViz
Visualization

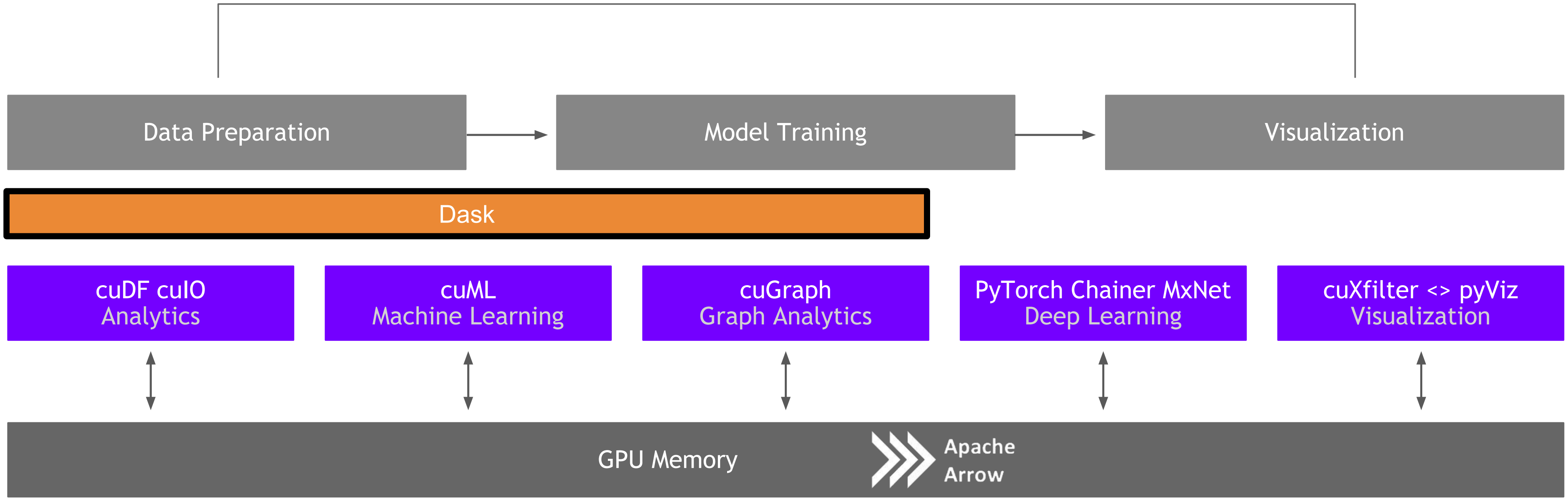
GPU Memory

Apache
Arrow

Dask

RAPIDS

Scaling RAPIDS with Dask



Why Dask?

PyData Native

- **Easy Migration:** Built on top of NumPy, Pandas Scikit-Learn, etc.
- **Easy Training:** With the same APIs
- **Trusted:** With the same developer community

Deployable

- **HPC:** SLURM, PBS, LSF, SGE
- **Cloud:** Kubernetes
- **Hadoop/Spark:** Yarn

DASK

Easy Scalability

- Easy to install and use on a laptop
- Scales out to thousand-node clusters

Popular

- Most common parallelism framework today in the PyData and SciPy community

Why OpenUCX?

Bringing hardware accelerated communications to Dask

- TCP sockets are slow!
- UCX provides uniform access to transports (TCP, InfiniBand, shared memory, NVLink)
- Python bindings for UCX (ucx-py) in the works
- Will provide best communication performance, to Dask based on available hardware on nodes/cluster

Scale up with RAPIDS

Scale Up / Accelerate

RAPIDS and Others

Accelerated on single GPU

NumPy -> CuPy/PyTorch/..
Pandas -> cuDF
Scikit-Learn -> cuML
Numba -> Numba

PyData

NumPy, Pandas, Scikit-Learn,
Numba and many more

Single CPU core
In-memory data

Scale out with RAPIDS + Dask with OpenUCX

Scale Up / Accelerate

RAPIDS and Others

Accelerated on single GPU

NumPy -> CuPy/PyTorch/..
Pandas -> cuDF
Scikit-Learn -> cuML
Numba -> Numba

RAPIDS + Dask with OpenUCX

Multi-GPU
On single Node (DGX)
Or across a cluster

PyData

NumPy, Pandas, Scikit-Learn, Numba and many more

Single CPU core
In-memory data

Dask

Multi-core and Distributed PyData

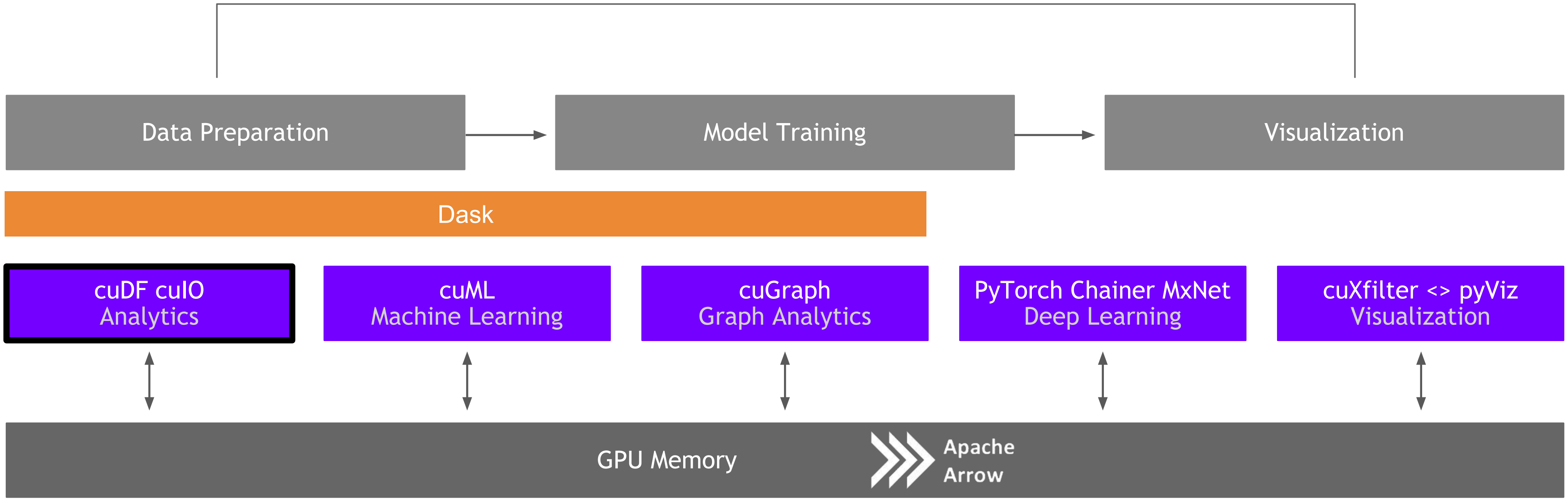
NumPy -> Dask Array
Pandas -> Dask DataFrame
Scikit-Learn -> Dask-ML
… -> Dask Futures

Scale out / Parallelize

cuDF

RAPIDS

GPU Accelerated data wrangling and feature engineering



ETL - the Backbone of Data Science

libcuDF is...

CUDA C++ Library

- Low level library containing function implementations and C/C++ API
- Importing/exporting Apache Arrow in GPU memory using CUDA IPC
- CUDA kernels to perform element-wise math operations on GPU DataFrame columns
- CUDA sort, join, groupby, reduction, etc. operations on GPU DataFrames

```
void some_function( cudf::column const* input,
                    cudf::column * output,
                    args...)
{
    // Do something with input
    // Produce output
}
```

ETL - the Backbone of Data Science

cuDF is…

```
In [2]: #Read in the data. Notice how it decompresses as it reads the data into memory.
        gdf = cudf.read_csv('/rapids/Data/black-friday.zip')

In [3]: #Taking a look at the data. We use "to_pandas()" to get the pretty printing.
        gdf.head().to_pandas()
```

Out[3]:

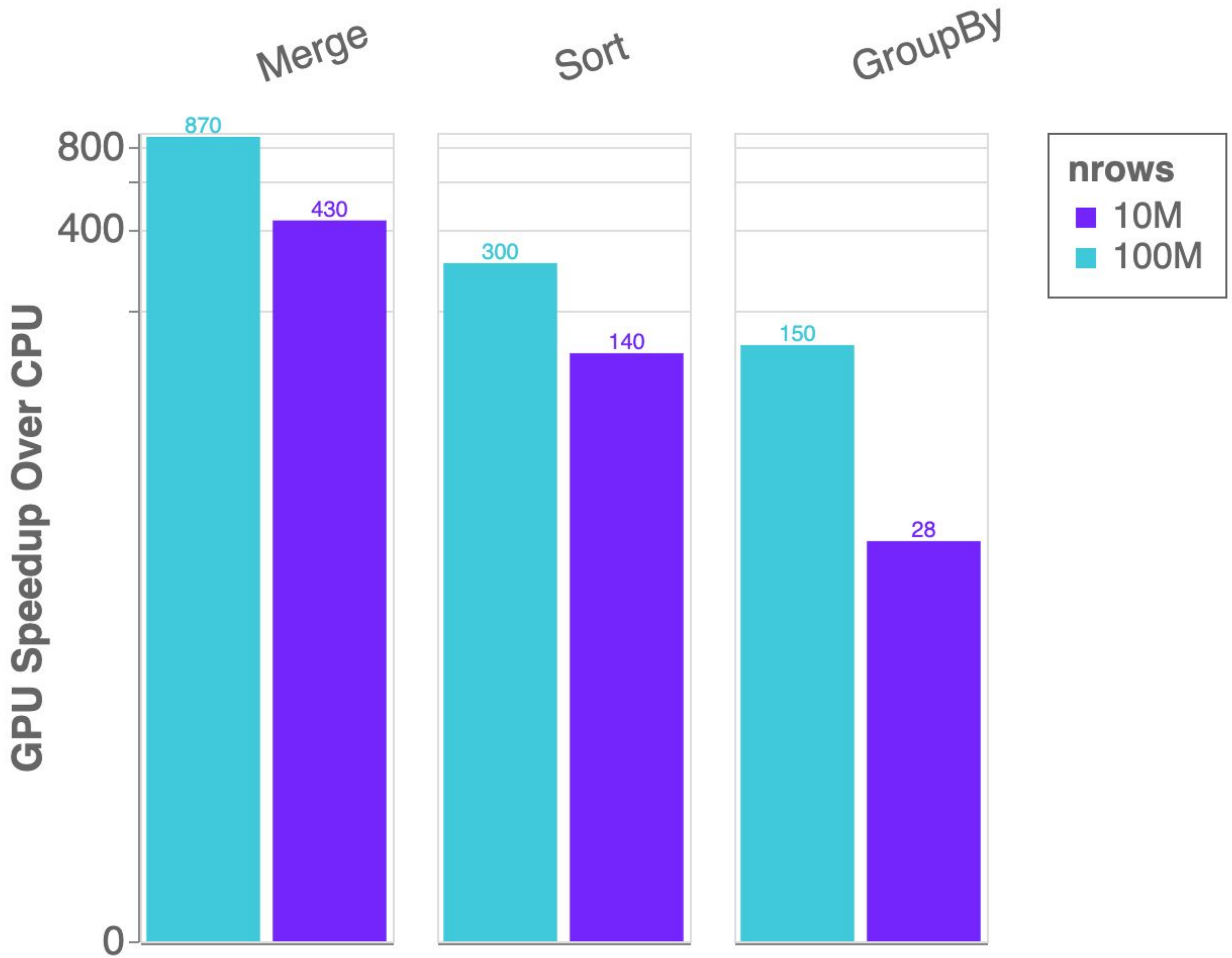
	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Cat
0	1000001	P00069042	F	0-17	10	A	2	0	3
1	1000001	P00248942	F	0-17	10	A	2	0	1
2	1000001	P00087842	F	0-17	10	A	2	0	12
3	1000001	P00085442	F	0-17	10	A	2	0	12
4	1000002	P00285442	M	55+	16	C	4+	0	8

```
In [6]: #grabbing the first character of the years in city string to get rid of plus sign, and converting
         to int
        gdf['city_years'] = gdf.Stay_In_Current_City_Years.str.get(0).stoi()

In [7]: #Here we can see how we can control what the value of our dummies with the replace method and turn
        strings to ints
        gdf['City_Category'] = gdf.City_Category.str.replace('A', '1')
        gdf['City_Category'] = gdf.City_Category.str.replace('B', '2')
        gdf['City_Category'] = gdf.City_Category.str.replace('C', '3')
        gdf['City_Category'] = gdf['City_Category'].str.stoi()
```

Python Library

- A Python library for manipulating GPU DataFrames following the Pandas API
- Python interface to CUDA C++ library with additional functionality
- Creating GPU DataFrames from Numpy arrays, Pandas DataFrames, and PyArrow Tables
- JIT compilation of User-Defined Functions (UDFs) using Numba

Benchmarks: single-GPU Speedup vs. Pandas



cuDF v0.9, Pandas 0.24.2

Running on NVIDIA DGX-1:

GPU: NVIDIA Tesla V100 32GB
CPU: Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

Benchmark Setup:

DataFrames: 2x int32 columns key columns, 3x int32 value columns

Merge: inner

GroupBy: count, sum, min, max calculated for each value column

ETL - the Backbone of Data Science

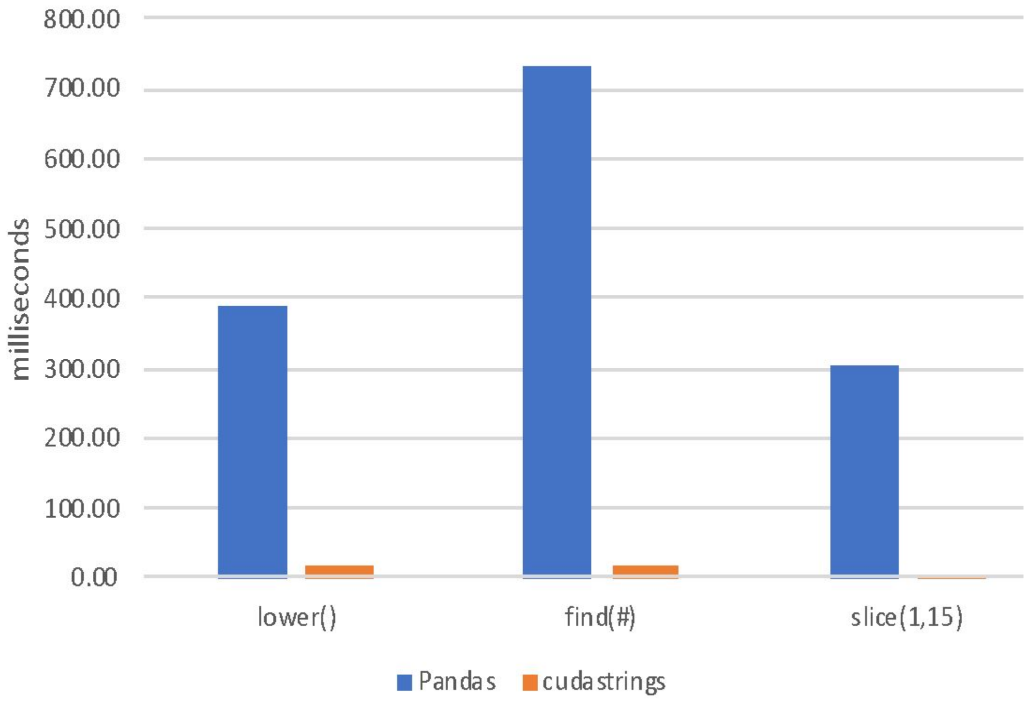
String Support

Current v0.9 String Support

- Regular Expressions
- Element-wise operations
  - Split, Find, Extract, Cat, Typecasting, etc…
- String GroupBys, Joins
- Categorical columns fully on GPU

Future v0.10+ String Support

- Combining cuStrings into libcudf
- Extensive performance optimization
- More Pandas String API compatibility
- JIT-compiled String UDFs



Extraction is the Cornerstone

cuIO for Faster Data Loading

- Follow Pandas APIs and provide >10x speedup

- CSV Reader - v0.2, CSV Writer v0.8
- Parquet Reader – v0.7, Parquet Writer v0.10
- ORC Reader – v0.7, ORC Writer v0.10
- JSON Reader - v0.8
- Avro Reader - v0.9

- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!

- Key is GPU-accelerating both parsing and decompression wherever possible

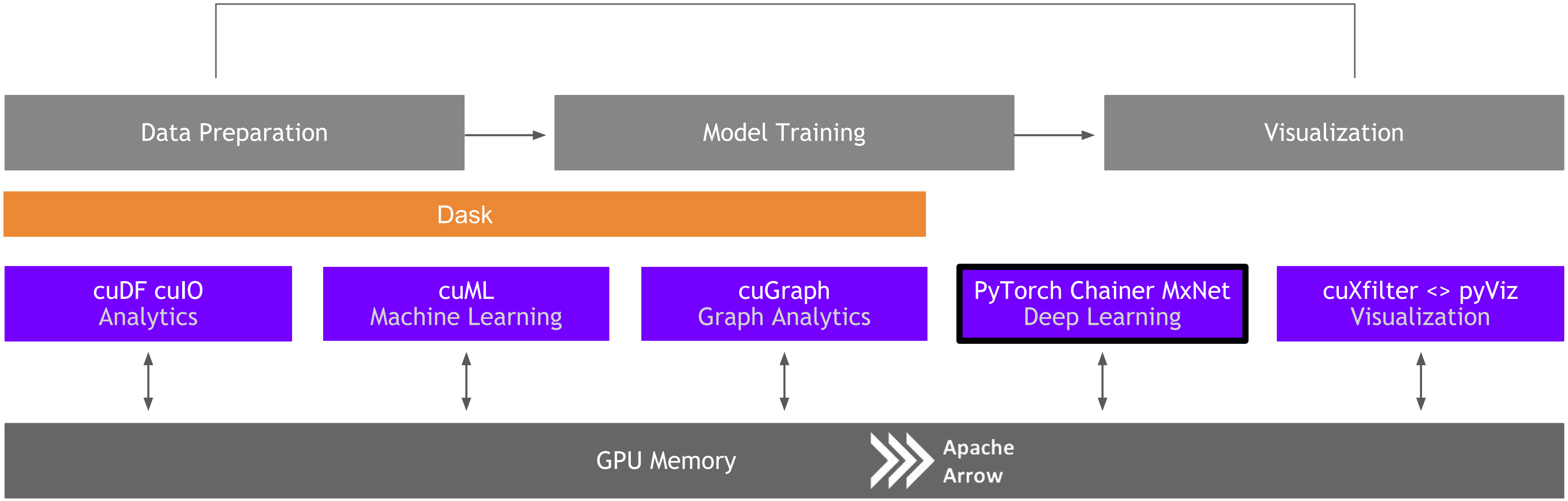
```
1]: import pandas, cudf

2]: %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
Wall time: 29.2 s
2]: 12748986

3]: %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
Wall time: 2.12 s
3]: 12748986

4]: !du -hs data/nyc/yellow_tripdata_2015-01.csv
1.9G    data/nyc/yellow_tripdata_2015-01.csv
```

Source: Apache Crail blog: SQL Performance: Part 1 - Input File Formats

ETL is not just DataFrames!

RAPIDS

Building bridges into the array ecosystem



Interoperability for the Win

DLPack and __cuda_array_interface__

PYTORCH

mpi4py

mxnet

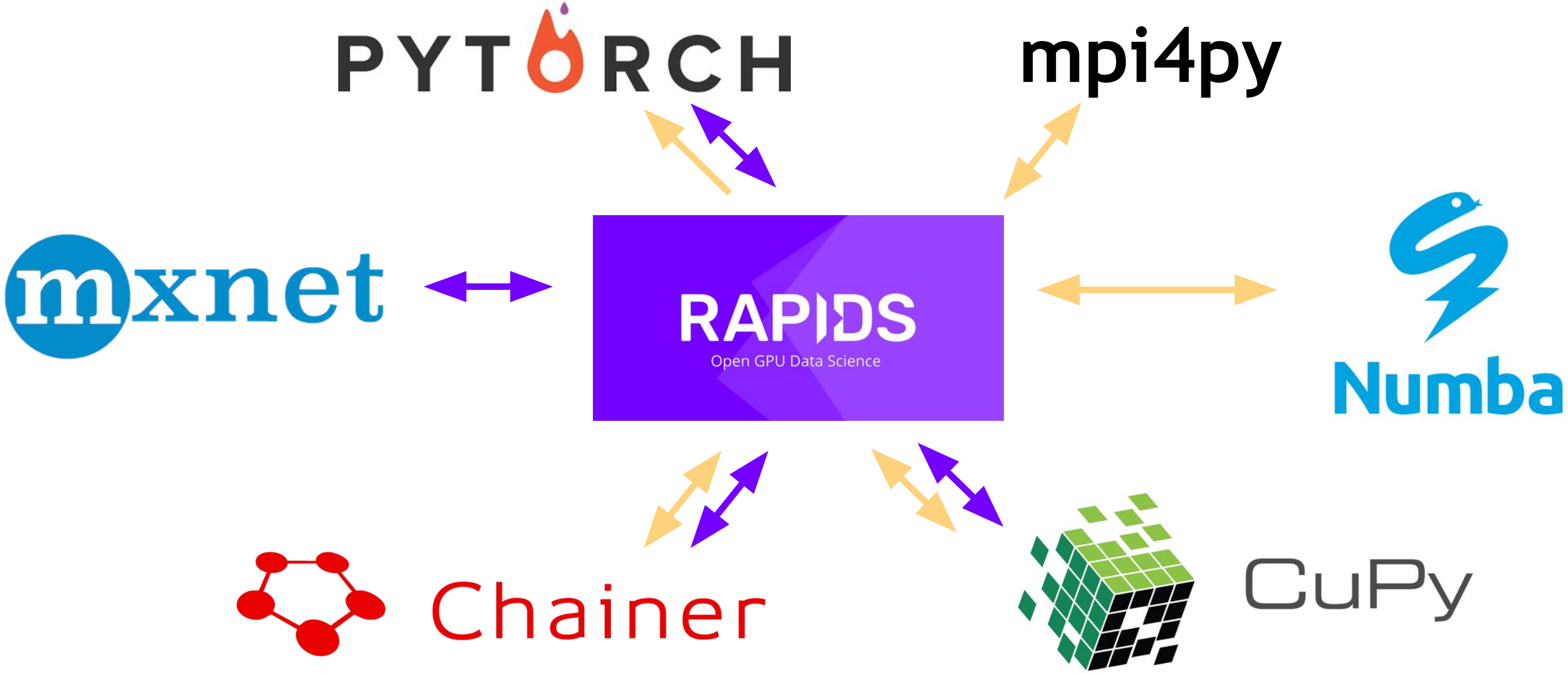
Numba

Chainer

CuPy

Interoperability for the Win

DLPack and __cuda_array_interface__



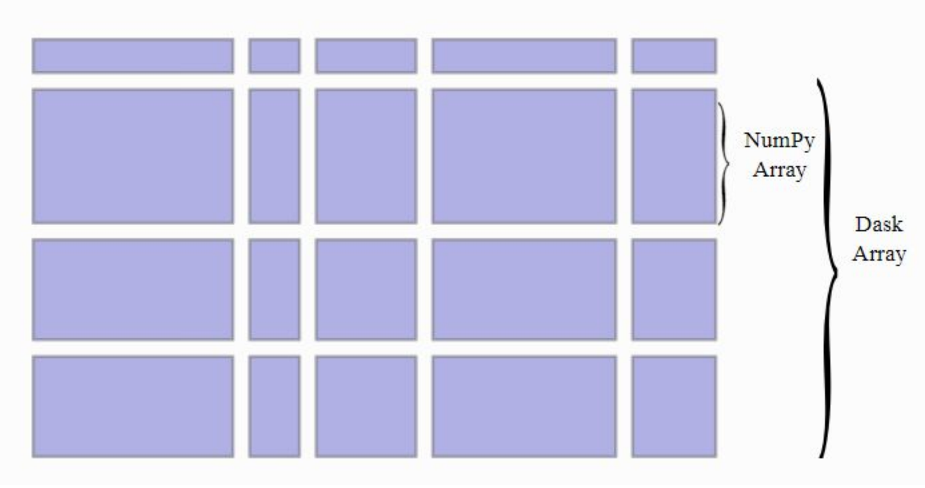
ETL – Arrays and DataFrames

Dask and CUDA Python arrays

Chainer

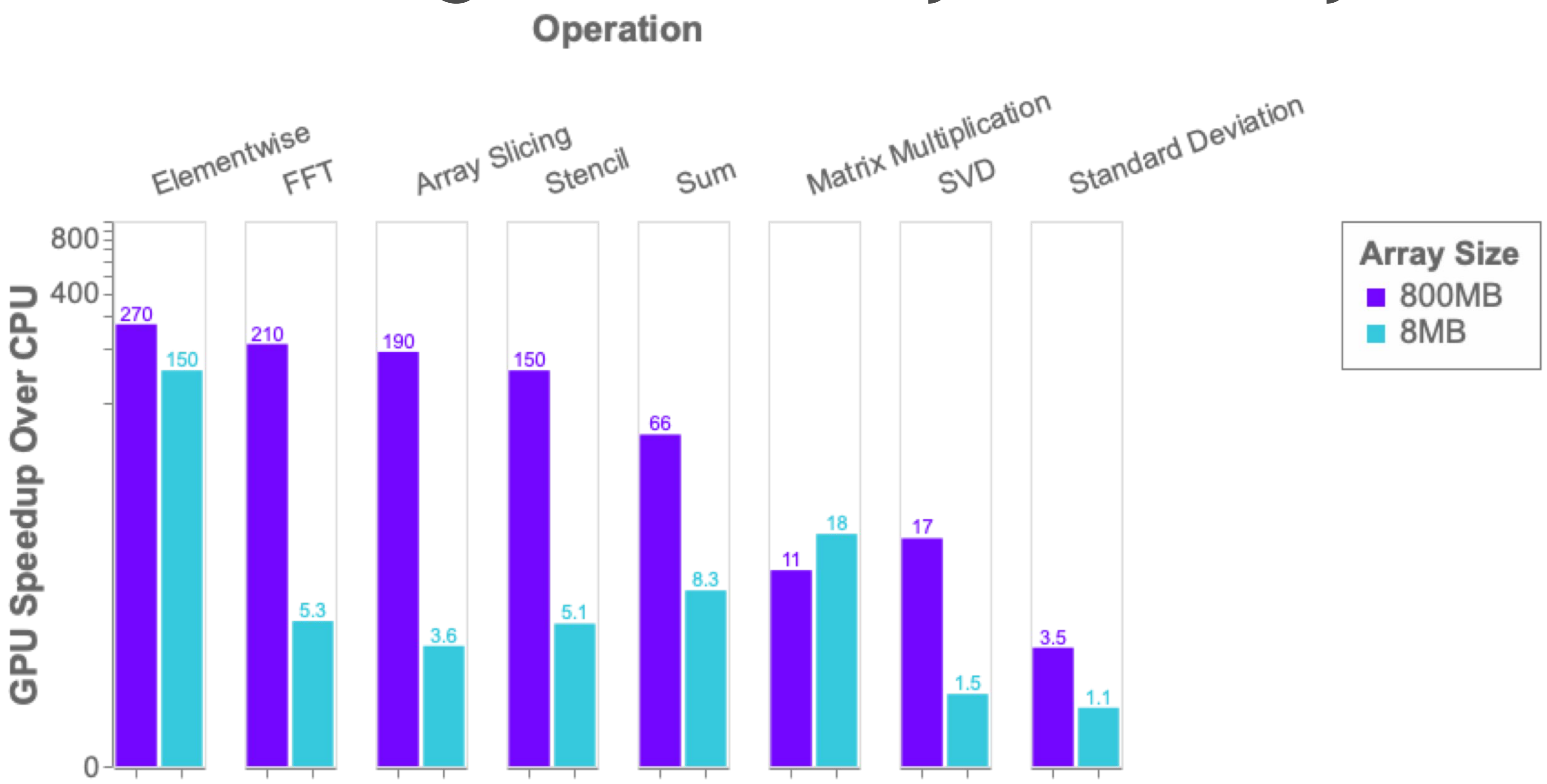
CuPy

Numba



- Scales NumPy to distributed clusters
- Used in climate science, imaging, HPC analysis up to 100TB size
- Now seamlessly accelerated with GPUs

Benchmark: single-GPU CuPy vs NumPy



More details: https://blog.dask.org/2019/06/27/single-gpu-cupy-benchmarks

Also…Achievement Unlocked:

Petabyte Scale Data Analytics with Dask and CuPy

Architecture	Time
Single CPU Core	2hr 39min
Forty CPU Cores	11min 30s
One GPU	1min 37s
Eight GPUs	19s

https://blog.dask.org/2019/01/03/dask-array-gpus-first-steps

3.2 PETABYTES IN LESS THAN 1 HOUR

Distributed GPU array | parallel reduction | using 76x GPUs

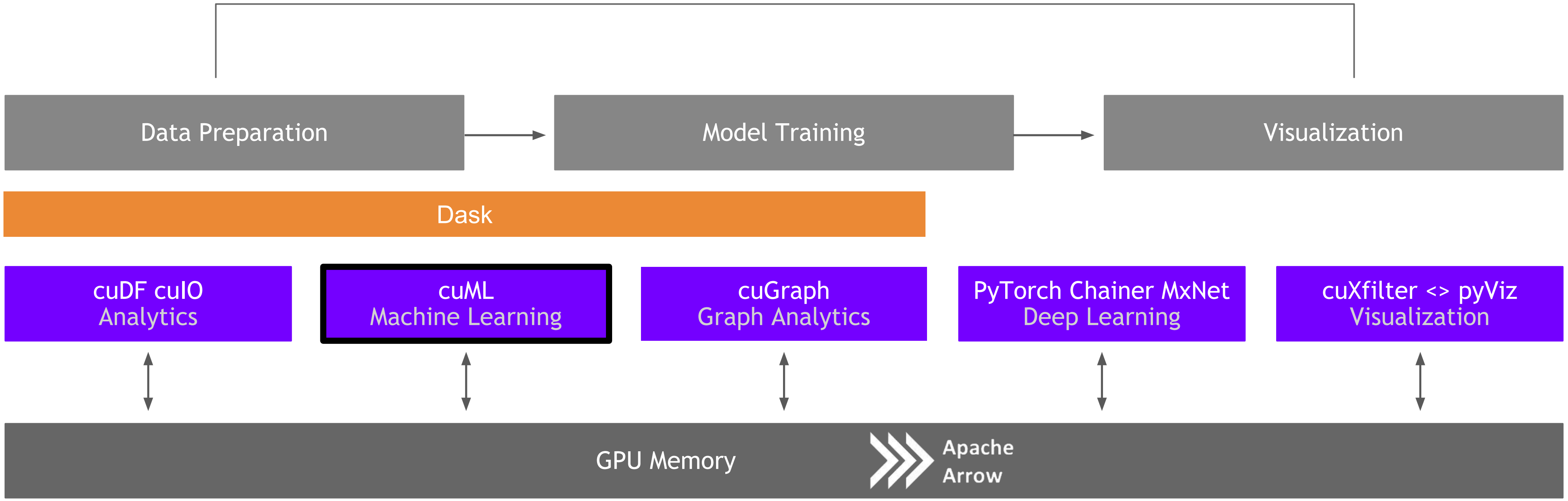
Array size	Wall Time (data creation + compute)
3.2 PB (20M x 20M doubles)	54 min 51 s

Cluster configuration: 20x GCP instances, each instance has:
CPU: 1 VM socket (Intel Xeon CPU @ 2.30GHz), 2-core, 2 threads/core, 132GB mem, GbE ethernet, 950 GB disk
GPU: 4x NVIDIA Tesla P100-16GB-PCIe (total GPU DRAM across nodes 1.22 TB)
Software: Ubuntu 18.04, RAPIDS 0.5.1, Dask=1.1.1, Dask-Distributed=1.1.1, CuPY=5.2.0, CUDA 10.0.130

cuML

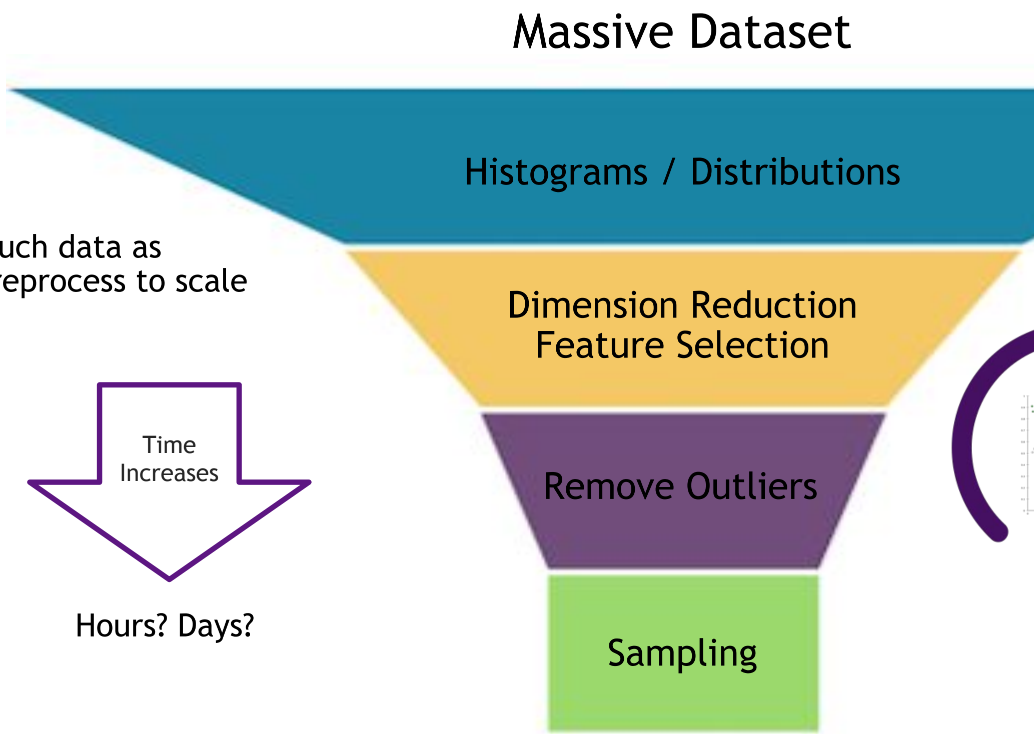
Machine Learning

More models more problems



Problem

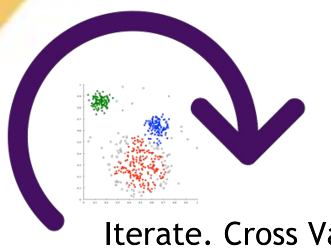
Data sizes continue to grow



Better to start with as much data as possible and explore / preprocess to scale to performance needs.

Time Increases

Hours? Days?

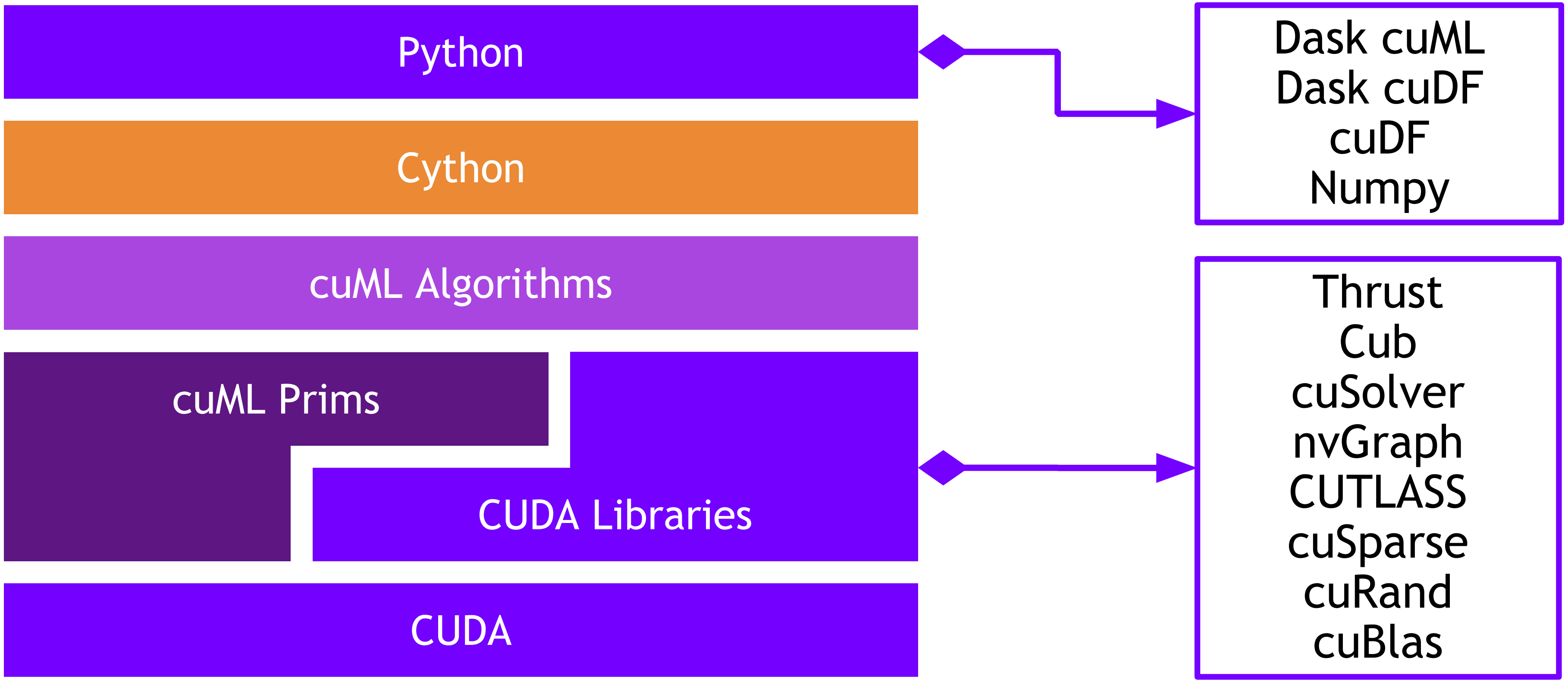


Iterate. Cross Validate & Grid Search.

Iterate some more.

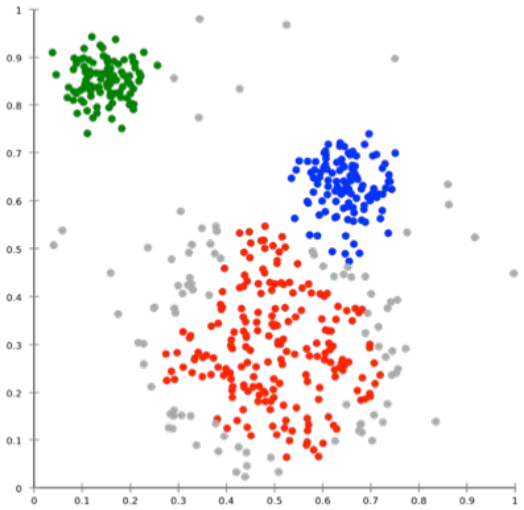
Meet reasonable speed vs accuracy tradeoff

ML Technology Stack



Algorithms

GPU-accelerated Scikit-Learn



Cross Validation

Hyper-parameter Tuning

More to come!

Classification / Regression

- Decision Trees / Random Forests
- Linear Regression
- Logistic Regression
- K-Nearest Neighbors

Inference

- **Random forest / GBDT inference**

Clustering

- K-Means
- DBSCAN
- Spectral Clustering

Decomposition & Dimensionality Reduction

- Principal Components
- Singular Value Decomposition
- UMAP
- Spectral Embedding

Time Series

- **Holt-Winters**
- Kalman Filtering

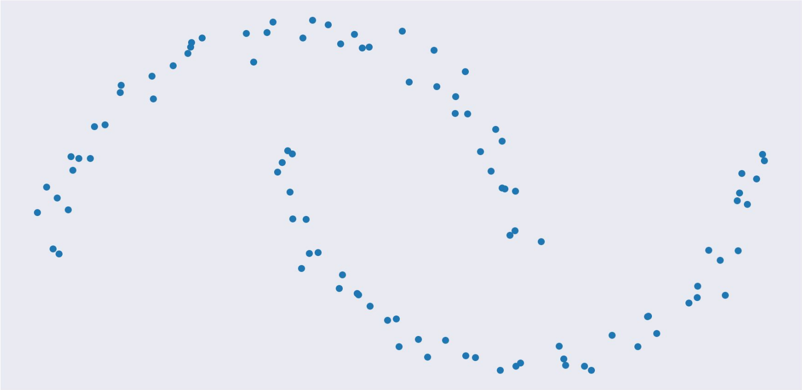
Key:

- Preexisting
- **NEW for 0.9**

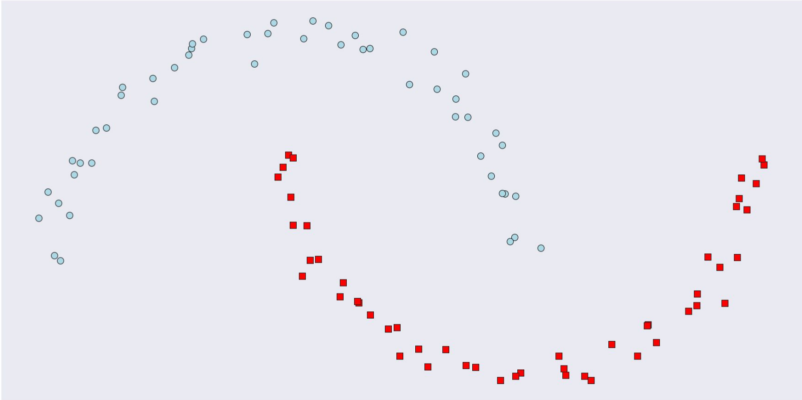
RAPIDS matches common Python APIs

CPU-Based Clustering

```
from sklearn.datasets import make_moons
import pandas

X, y = make_moons(n_samples=int(1e2),
                  noise=0.05, random_state=0)

X = pandas.DataFrame({'fea%d'%i: X[:, i]
                     for i in range(X.shape[1])})
```



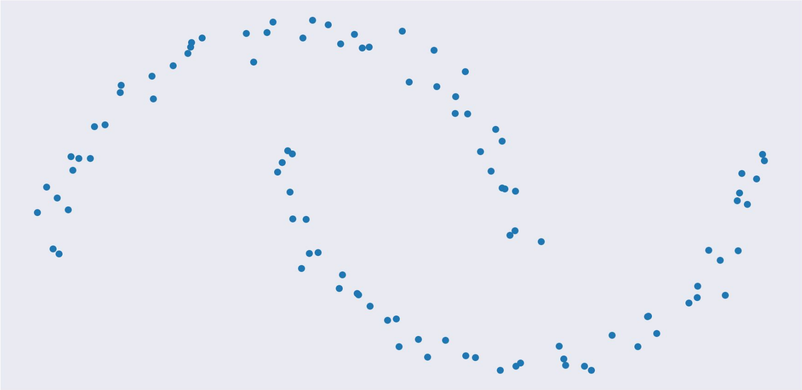
```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)

dbscan.fit(X)

y_hat = dbscan.predict(X)
```



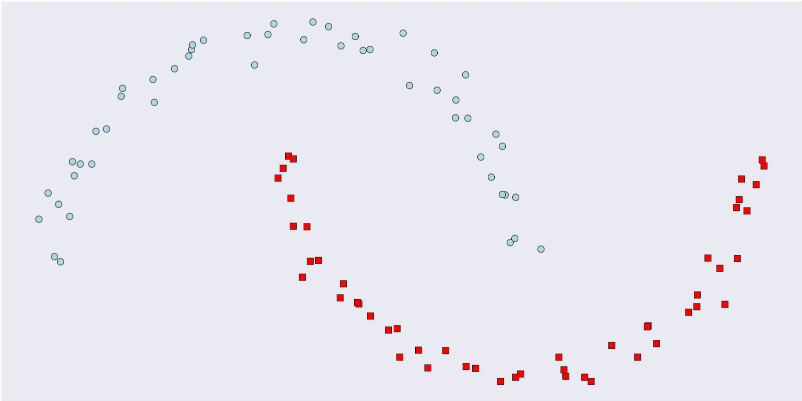
RAPIDS matches common Python APIs

GPU-Accelerated Clustering

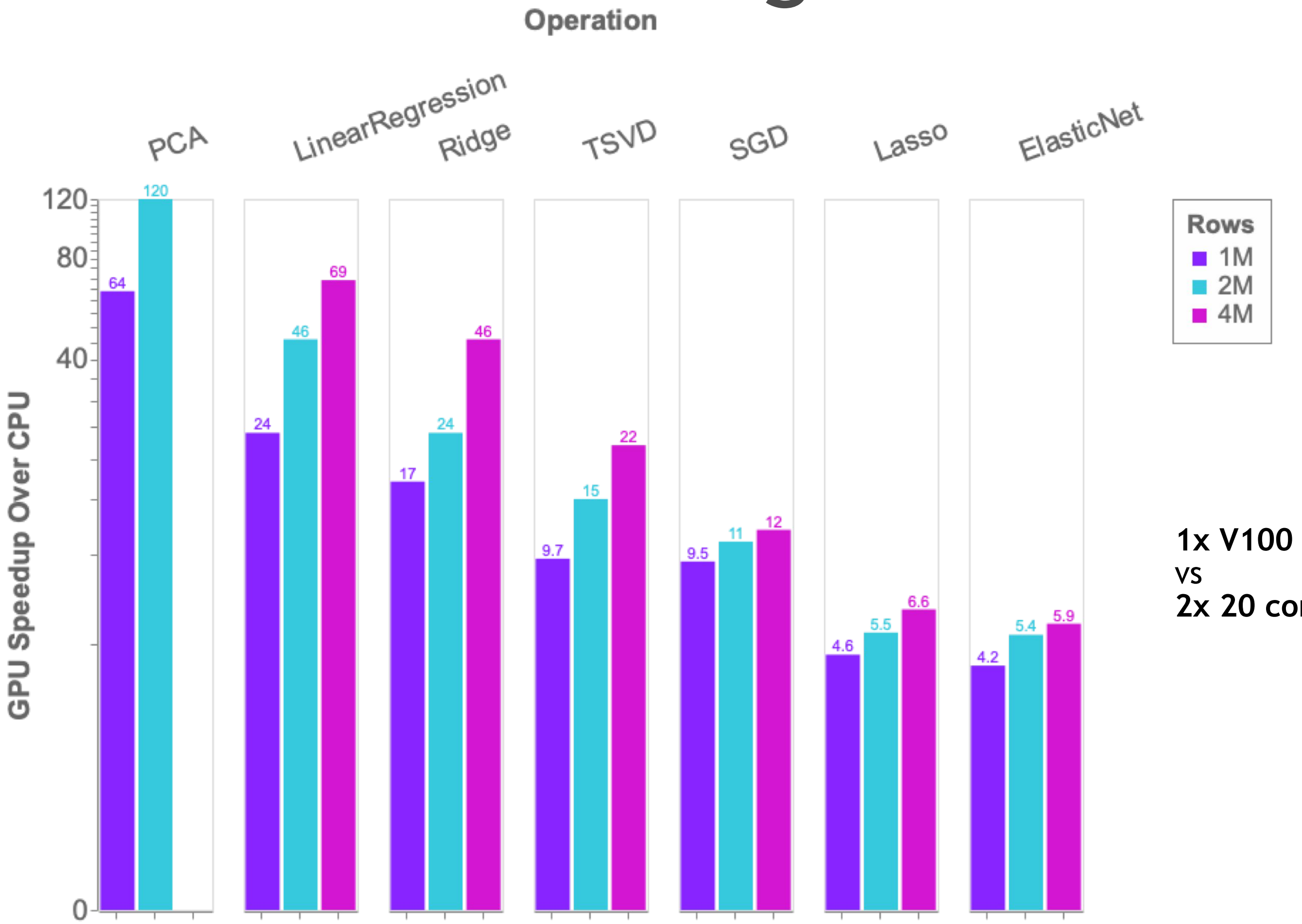
```
from sklearn.datasets import make_moons
import cudf

X, y = make_moons(n_samples=int(1e2),
                  noise=0.05, random_state=0)

X =   cudf.DataFrame({'fea%d'%i: X[:, i]
                     for i in range(X.shape[1])})
```



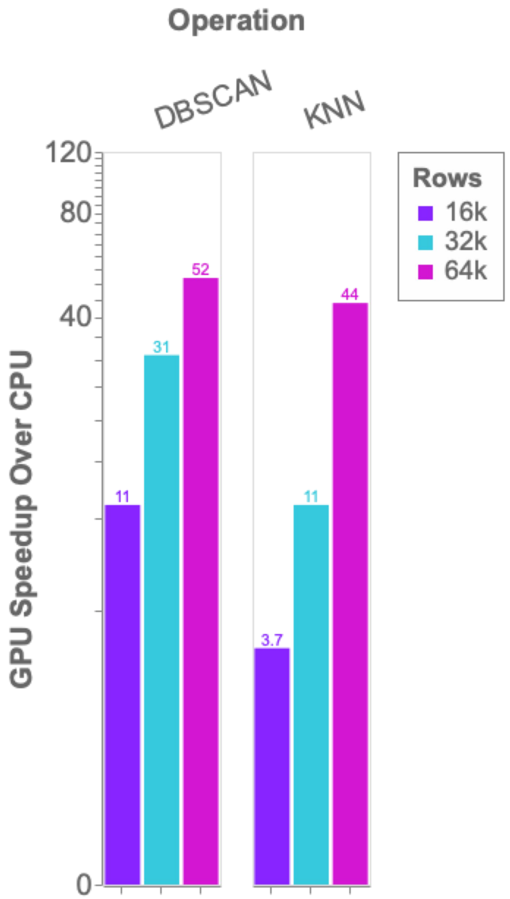
```
from cuml              import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)

dbscan.fit(X)

y_hat = dbscan.predict(X)
```



Benchmarks: single-GPU cuML vs scikit-learn



1x V100
vs
2x 20 core CPU



Road to 1.0

August 2019 - RAPIDS 0.9

cuML	Single-GPU	Multi-GPU	Multi-Node-Multi-GPU
Gradient Boosted Decision Trees (GBDT)			
GLM			
Logistic Regression			
Random Forest			
K-Means			
K-NN			
DBSCAN			
UMAP			
Holt-Winters			
Kalman Filter			
t-SNE			
Principal Components			
Singular Value Decomposition			

Road to 1.0

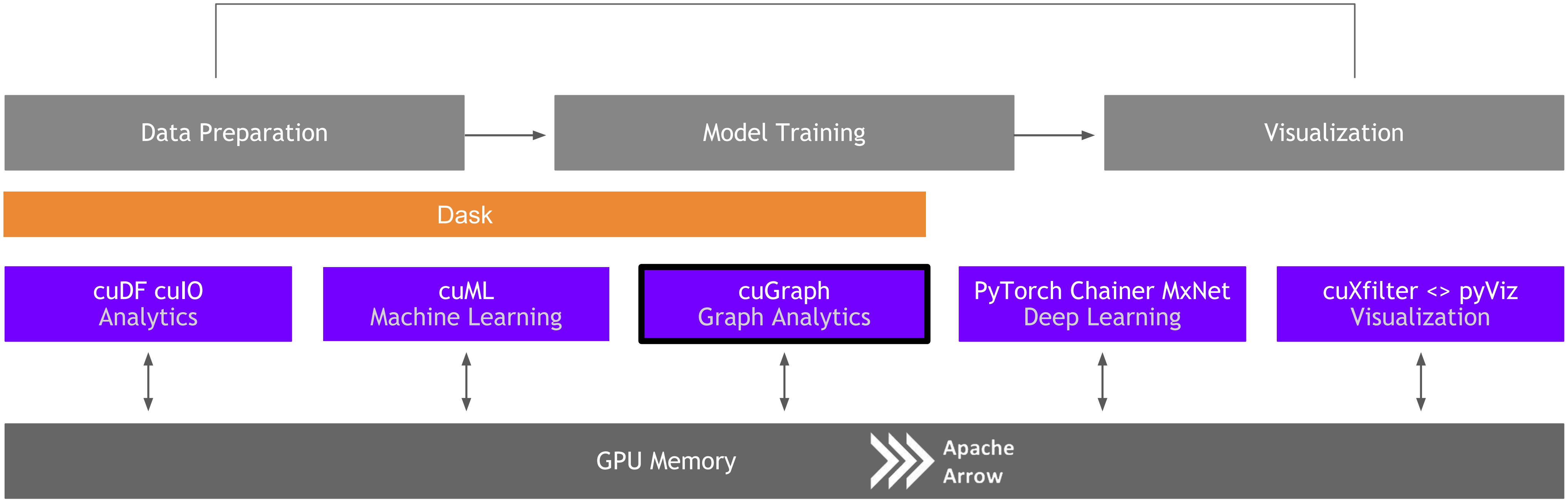
March 2020 - RAPIDS 0.14

cuML	Single-GPU	Multi-GPU	Multi-Node-Multi-GPU
Gradient Boosted Decision Trees (GBDT)			
GLM			
Logistic Regression			
Random Forest			
K-Means			
K-NN			
DBSCAN			
UMAP			
ARIMA & Holt-Winters			
Kalman Filter			
t-SNE			
Principal Components			
Singular Value Decomposition			

cuGraph

Graph Analytics

More connections more insights



GOALS AND BENEFITS OF CUGRAPH

Focus on Features and User Experience

Breakthrough Performance

- Up to 500 million edges on a single 32GB GPU
- Multi-GPU support for scaling into the billions of edges

Multiple APIs

- **Python:** Familiar NetworkX-like API
- **C/C++:** lower-level granular control for application developers

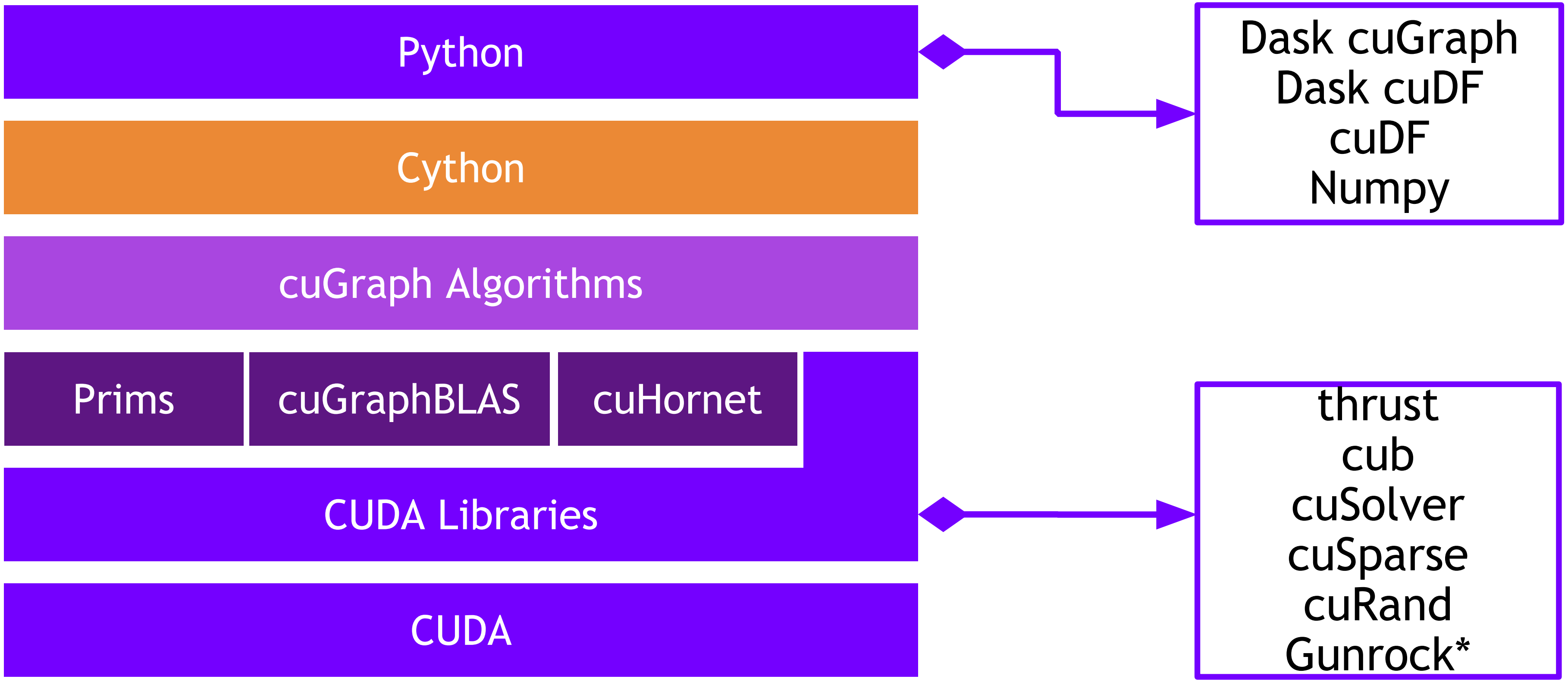
Seamless Integration with cuDF and cuML

- Property Graph support via DataFrames

Growing Functionality

- Extensive collection of algorithm, primitive, and utility functions

Graph Technology Stack

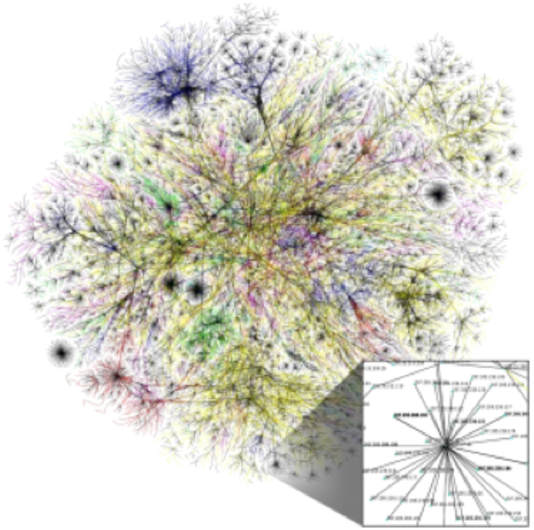


nvGRAPH has been Opened Sourced and integrated into cuGraph. A legacy version is available in a RAPIDS GitHub repo

* Gunrock is from UC Davis

Algorithms

GPU-accelerated NetworkX



Query Language

Multi-GPU

Utilities

More to come!

Community	Spectral Clustering Balanced-Cut Modularity Maximization Louvain Subgraph Extraction Triangle Counting
Components	Weakly Connected Components **Strongly Connected Components**
Link Analysis	Page Rank (**Multi-GPU**) Personal Page Rank
Link Prediction	Jaccard Weighted Jaccard Overlap Coefficient
Traversal	Single Source Shortest Path (SSSP) Breadth First Search (BFS)
Structure	COO-to-CSR (**Multi-GPU**) Transpose Renumbering

Louvain Single Run

```
G = cugraph.Graph()
G.add_edge_list(gdf["src_0"], gdf["dst_0"], gdf["data"])
df, mod = cugraph.nvLouvain(G)
```

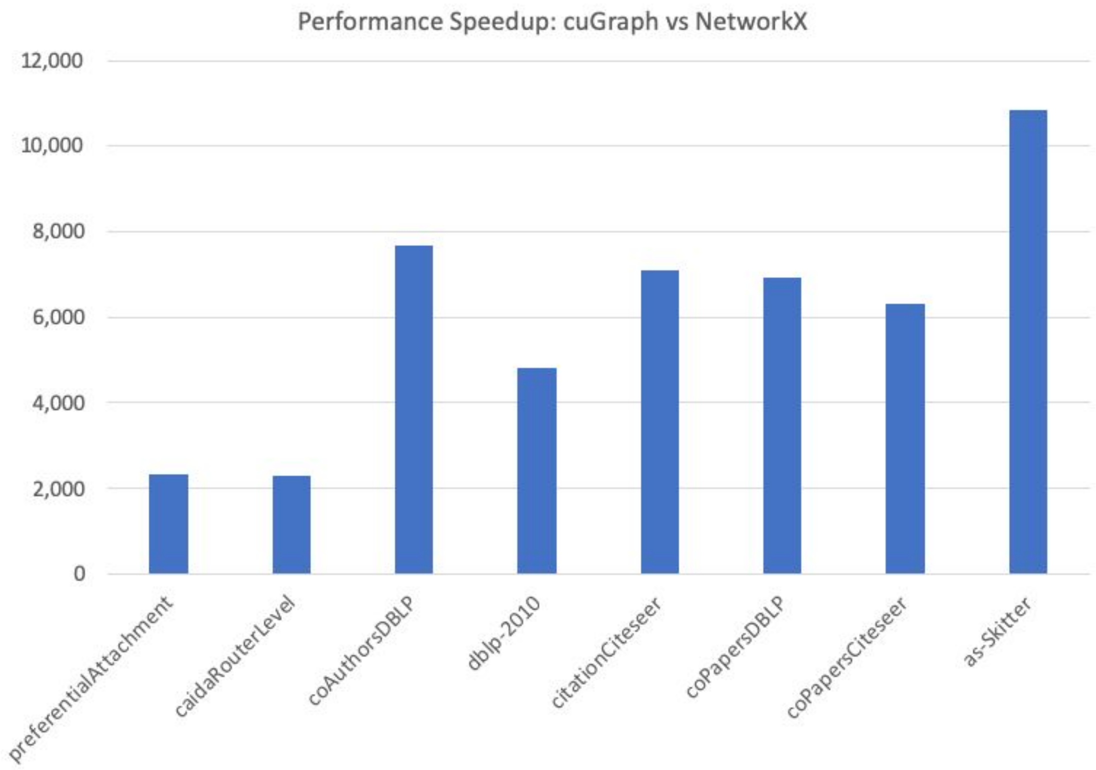
Louvain returns:

cudf.DataFrame with two names columns:

louvain["vertex"]: The vertex id.

louvain["partition"]: The assigned partition.

Dataset	Nodes	Edges
preferentialAttachment	100,000	999,970
caidaRouterLevel	192,244	1,218,132
coAuthorsDBLP	299,067	299,067
dblp-2010	326,186	1,615,400
citationCiteseer	268,495	2,313,294
coPapersDBLP	540,486	30,491,458
coPapersCiteseer	434,102	32,073,440
as-Skitter	1,696,415	22,190,596



Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

HiBench Scale	Vertices	Edges	CSV File (GB)	# of GPUs	PageRank for 3 Iterations (secs)
Huge	5,000,000	198,000,000	3	1	1.1
BigData	50,000,000	1,980,000,000	34	3	5.1
BigData x2	100,000,000	4,000,000,000	69	6	9.0
BigData x4	200,000,000	8,000,000,000	146	12	18.2
BigData x8	400,000,000	16,000,000,000	300	16	31.8

Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

HiBench Scale	Vertices	Edges	CSV File (GB)	# of GPUs	PageRank for 3 Iterations (secs)
Huge	5,000,000	198,000,000	3	1	1.1
BigData	50,000,000	1,980,000,000	34	3	5.1
BigData x2	100,000,000	4,000,000,000	69	6	9.0
BigData x4	200,000,000	8,000,000,000	146	12	18.2
BigData x8	400,000,000	16,000,000,000	300	16	31.8

BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX ⇒ 96 mins!

Road to 1.0

August 2019 - RAPIDS 0.9

cuGraph	Single-GPU	Multi-GPU	Multi-Node-Multi-GPU
Jaccard and Weighted Jaccard			
Page Rank			
Personal Page Rank			
SSSP			
BFS			
Triangle Counting			
Subgraph Extraction			
Katz Centrality			
Betweenness Centrality			
Connected Components (Weak and Strong)			
Louvain			
Spectral Clustering			
InfoMap			
K-Cores			

Road to 1.0

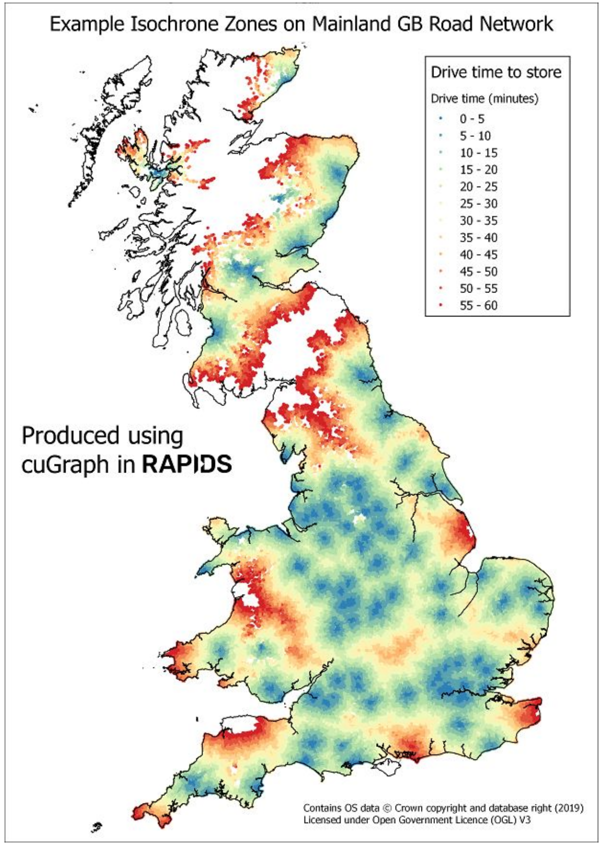
March 2020 - RAPIDS 0.14

cuGraph	Single-GPU	Multi-GPU	Multi-Node-Multi-GPU
Jaccard and Weighted Jaccard			
Page Rank			
Personal Page Rank			
SSSP			
BFS			
Triangle Counting			
Subgraph Extraction			
Katz Centrality			
Betweenness Centrality			
Connected Components (Weak and Strong)			
Louvain			
Spectral Clustering			
InfoMap			
K-Cores			

RAPIDS Geospatial Applications

RAPIDS Geospatial Applications

cuGraph SSSP



Read the source data and adjust the vertex IDs

```python
# Import needed libraries
import cugraph
import cudf
import numpy as np
```

```python
# Test file  - Read OS Open Roads as a graph. (Store dataset in ./data folder)
datafile='/data/road_graph/gb_road_graph_20190528.csv'
```

```python
# Read the data file with length, drive time and coordinates
cols = ["src", "dst","length","drivetime","type","x1","y1","x2","y2"]
dtypes =["int32", "int32", "float32", "float32", "int32", "float64", "float64", "float64", "float64"]
gdf = cudf.read_csv(datafile, names=cols, dtype=dtypes ,skiprows=1)
```

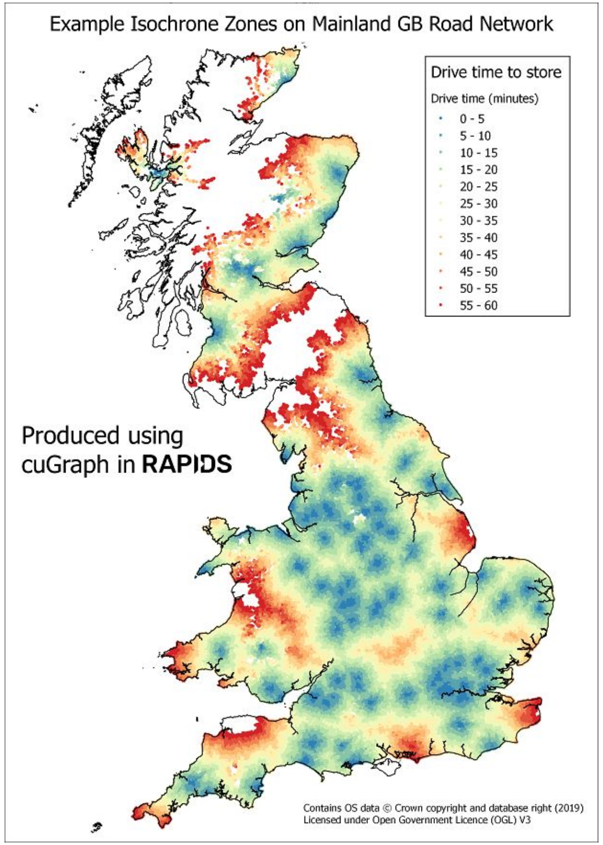
```python
# Need to shift the vertex IDs to start with zero rather than one (next version of cuGraph will fix this issue)
gdf["src_0"] = gdf["src"] - 1
gdf["dst_0"] = gdf["dst"] - 1
```

```python
# Display the results
print(gdf)
```

```
   src  dst  length  drivetime  type        x1         y1 ...  dst_0
0   16   20   162.0  18.119202    37  464331.0  1213440.0 ...     19
1   20   16   162.0  18.119202    37  464387.0  1213370.0 ...     15
2   56   55   855.0     95.629    37  462043.0  1213380.0 ...     54
3   55   56   855.0     95.629    37  461949.0  1214150.0 ...     55
4   72   22   382.0    17.0902     9  464065.0  1213380.0 ...     21
5   22   72   382.0    17.0902     9  464433.0  1213480.0 ...     71
6   72   71   270.0    15.0993    25  464065.0  1213380.0 ...     70
7   71   72   270.0    15.0993    25  463833.0  1213520.0 ...     71
8   17   18    48.0    5.36865    37  464332.0  1213430.0 ...     17
9   18   17    48.0    5.36865    37  464333.0  1213390.0 ...     16
[7347796 more rows]
[3 more columns]
```

RAPIDS Geospatial Applications

cuGraph SSSP



Load the stores list and calculate one hour drive times

```
# Load stores list
datafile='/data/road_graph/test_stores.csv'
cols = ["id", "x","y"]
dtypes =["int32", "float64", "float64"]
sdf = cudf.read_csv(datafile, names=cols, dtype=dtypes ,skiprows=1)
stores = [sdf['id'].to_array(),sdf['x'].to_array(),sdf['y'].to_array()]
```

```
# Print stores list
print(sdf)
```

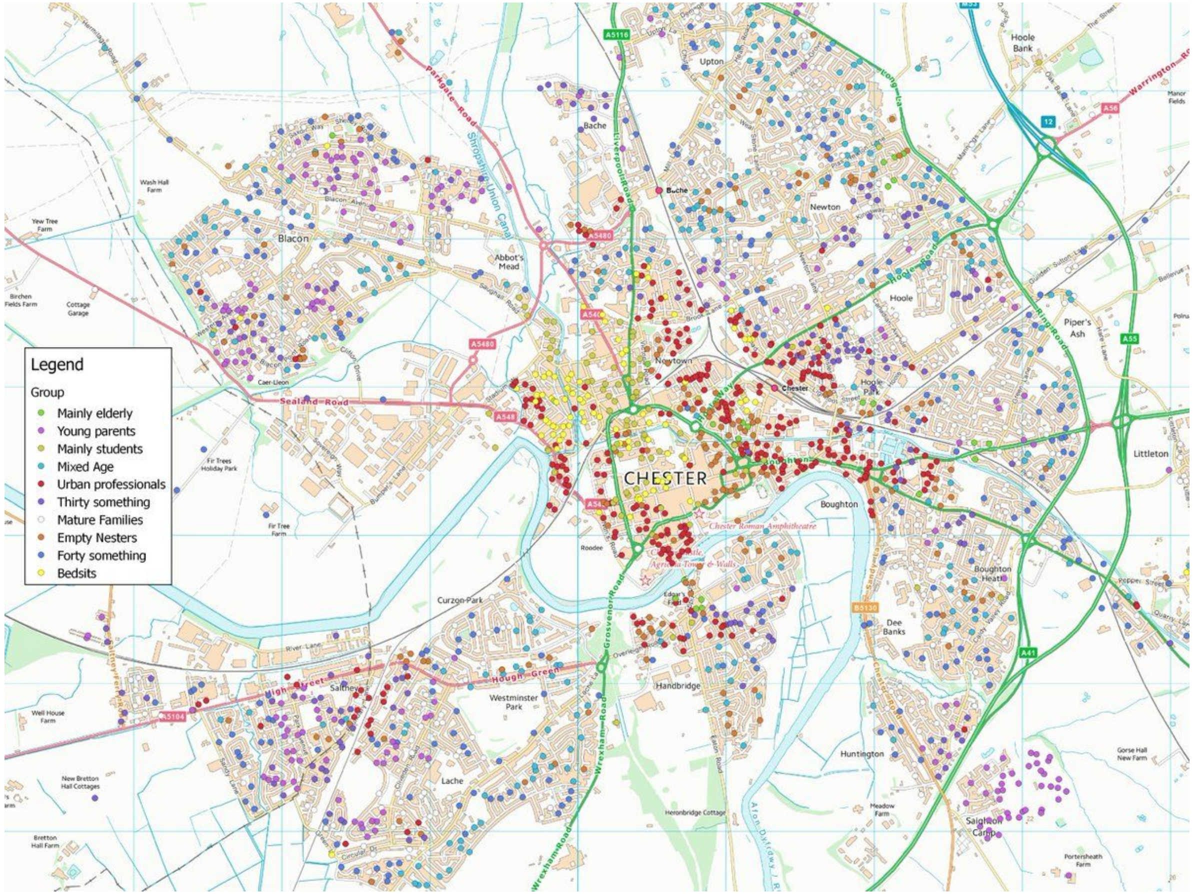
```
       id                  x                y
0   18257          324049.45        934680.08
1   22739          164087.59        823684.59
2   58323           388440.0         819496.0
3  102962           277367.0         701611.0
4  103272          263501.89  705534.7699999999
5  128342          339283.12        729293.71
6  145558  315452.83999999997        790942.41
7  146959  379043.98000000004        767064.31
8  275566          235338.62        623894.12
9  457087          448150.53        519823.61
[90 more rows]
```

```
%%time
store_drivetimes = []
# for each store find the drive times to all vertices within 1 hour drive time by filtering initially on 70 miles
for i in range(len(stores[0])):
    # initial filter on 70 miles (112654 metres) grid from store node
    query_x = "((x1>="+str(stores[1][i]-112654)+" and x1<="+str(stores[1][i]+112654)+") or (x2>="+str(stores[1][i]-112654)+" and x2<="+str(stores[1][i]+112654)+"))"
    query_y = "((y1>="+str(stores[2][i]-112654)+" and y1<="+str(stores[2][i]+112654)+") or (y2>="+str(stores[2][i]-112654)+" and y2<="+str(stores[2][i]+112654)+"))"
    qdf = gdf.query("("+query_x+" and "+query_y+")")
    # create a road Graph weighted on drive time
    G = cugraph.Graph()
    G.add_edge_list(qdf["src_0"], qdf["dst_0"],qdf['drivetime'])
    # Call cugraph.sssp to get the drivetime from store vertex
    df = cugraph.sssp(G,stores[0][i])
    # Filter 1 hour (3600 seconds) and save results
    store_drivetimes += [df.query("distance<=3600")]
```

```
CPU times: user 45.7 s, sys: 19.7 s, total: 1min 5s
Wall time: 1min 5s
```

RAPIDS Geospatial Applications

cuML K-means




```
[23]: # Pearson Correlation Coefficient
class stats_df(cudf.DataFrame):

    def pearson_correl(self,c1,c2):
        c1_m = self[c1].mean()
        c2_m = self[c2].mean()
        c1_s = self[c1].std()
        c2_s = self[c2].std()
        n = self[c1].count()
        return ((self[c1]*self[c2]).sum()-n*c1_m*c2_m) / ((n-1) * c1_s * c2_s)

    def prediction_interval(self,c):
        m = self[c].mean()
        s = self[c].std()
        return(m-s*1.96,m+s*1.96)
```

```
[24]: gdf.__class__ = stats_df
```

```
[25]: print(gdf.pearson_correl('P_INFANT','P_75+'))
print(np.array(gdf.prediction_interval('P_75+'))*(1-0.1357827525739096))

-0.503099498141717
[0.01640483 0.13712953]
```

```
[26]: m = (gdf['P_INFANT']*gdf['P_75+']).mean()
s = (gdf['P_INFANT']*gdf['P_75+']).std(ddof=1)
p = (gdf['P_INFANT']*gdf['P_75+']).std(ddof=0)
print(m,s,p)

0.002964567335037682 0.0010167776413762544 0.0010167773042338017
```

```
[27]: %%time
kids_kmeans = cuml.KMeans(n_clusters=10)
kids_kmeans.fit(gdf[fields])

CPU times: user 338 ms, sys: 74.9 ms, total: 413 ms
Wall time: 411 ms
```

RAPIDS Geospatial Applications

John Murray @MurrayData

"RAPIDS opens up new opportunities by simplifying the application of geographic data science at scale, at speed. Applications are limited only by your imagination.

"While we have achieved a lot with RAPIDS, in the short time since initial launch, I believe that we have only scratched the surface so far."

John Murray, Geographic Data Science Lab, University of Liverpool

RAPIDS Geospatial Applications

I’ll walk 500 miles...

It’s [every walkable road in Great Britain] a sizeable graph consisting of 3,078,131 vertices and 7,347,806 edges so represents a significant mathematical challenge, so I used Graphics Processing Unit (GPU) computing.

https://www.citymetric.com/horizons/so-where-exactly-did-proclaimers-walk-500-miles-4629

Geospatial Challenges

Still much more to do

Data Representation & Management

Indexing, Database Queries, Aggregation & KPIs

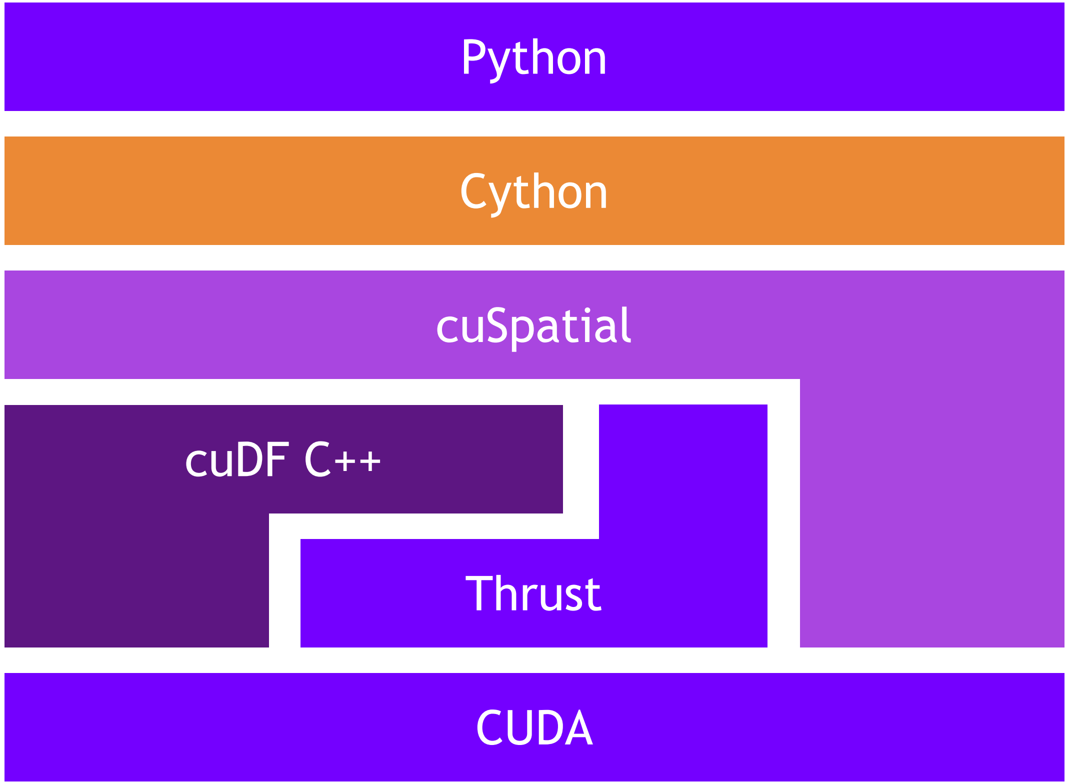
Positioning & Navigation (Indoor, Outdoor)

Machine Learning, Big Data Analytics, Behavior Models Event Analytics & Anomaly Detection

Map-based Visualization

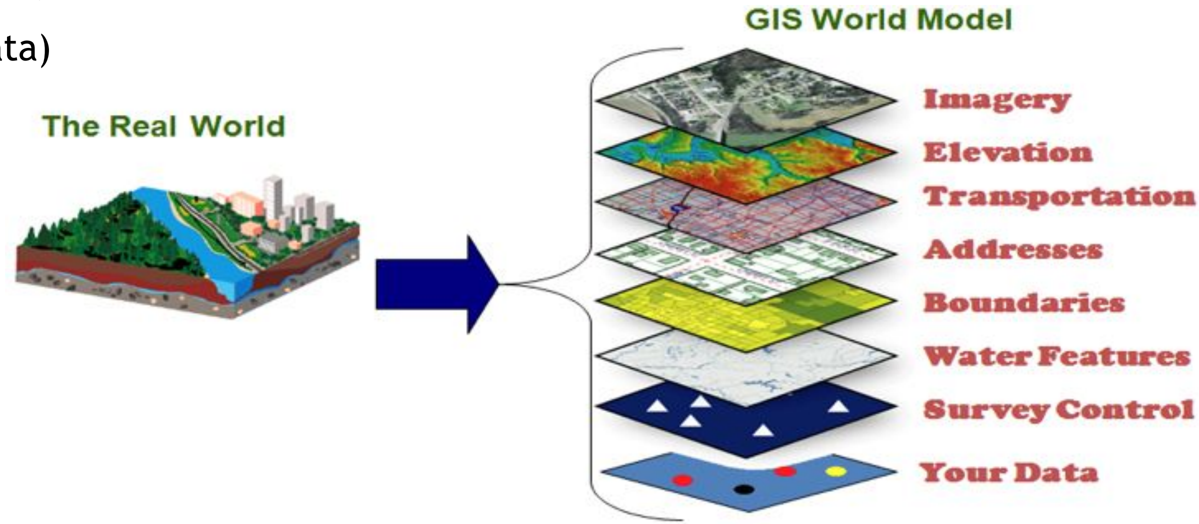
cuSpatial

cuSpatial Technology Stack



cuSpatial 0.10

1. Data Representation for point, line, polygon (Columinar/SoA)
2. Location Data Ingestion from JSON schema (IVA schema data)
3. **Spatial window query**
4. **Point-in-polygon test**
5. **Converting lat/lon to x/y**
6. **Haversine Distance between pairs of lat/lon points**
7. **Location-to-trajectory**
8. **Computing trajectory distance/speed**
9. **Computing trajectory spatial bounding box**
10. **Directed Hausdorff distance**
11. Python bindings for all the above features
12. Python test code, sample application & performance evaluation scripts



cuSpatial

Today and Tomorrow

Layer	0.10/0.11 Functionality	Functionality Roadmap (2020)
High-level Analytics	C++ Library w. Python bindings enabling distance, speed, trajectory similarity, trajectory clustering	C++ Library w. Python bindings for additional spatio-temporal trajectory clustering, acceleration, dwell-time, salient locations, trajectory anomaly detection, origin destination, etc.
Graph layer	cuGraph	Map matching, Djikstra algorithm, Routing
Query layer	Nearest Neighbor, Range Search	KNN, Spatiotemporal range search and joins
Index layer	Grid, Quad Tree	R-Tree, Geohash, Voronoi Tessellation
Geo-operations	Point in polygon (PIP), Haversine distance, Hausdorff distance, lat-lon to xy transformation	Line intersecting polygon, Other distance functions, Polygon intersection, union
Geo-representation	Shape primitives, points, polylines, polygons	Additional shape primitives

cuSpatial 0.10

Performance at a Glance

cuSpatial Operation	Input data	cuSpatial Runtime	Reference Runtime	Speedup
Point-in-Polygon Test	1.3+ million vehicle point locations and 27 Region of Interests	1.11 ms (C++) 1.50 ms (Python) [Nvidia Titan V]	334 ms (C++, optimized serial) 130468.2 ms (python Shapely API, serial) [Intel i7-7800X]	301X (C++) 86,978X (Python)
Haversine Distance Computation	13+ million Monthly NYC taxi trip pickup and drop-off locations	7.61 ms (Python) [Nvidia T4]	416.9 ms (Numba) [Nvidia T4]	54.7X (Python)
Hausdorff Distance Computation (for clustering)	10,700 trajectories with 1.3+ million points	13.5s [Quadro V100]	19227.5s (Python SciPy API, serial) [Intel i7-6700K]	1,400X (Python)

cuSpatial 0.10

Try It Today!

conda install -c rapidsai-nightly cuspatial

Community

Ecosystem Partners

CONTRIBUTORS

ANACONDA · BLAZINGDB · GUNROCK · NVIDIA · QUANSIGHT · Walmart

ADOPTERS

Chainer · databricks · graphistry · H2O.ai · IBM · iguazio · Inria
MAPR · omni sci · PyTorch · Uber · URSA LABS

OPEN SOURCE

Apache Arrow · DASK · GOAi · Numba · scikit learn · dmlc XGBoost

Building on top of RAPIDS

A bigger, better, stronger ecosystem for all

nuclio

blazingSQL

Streamz

High-Performance Serverless event and data processing that utilizes RAPIDS for GPU Acceleration

GPU accelerated SQL engine built on top of RAPIDS

Distributed stream processing using RAPIDS and Dask

Explore: RAPIDS Code and Blogs

Check out our code and how we use it

README.md

RAPIDS cuDF - GPU DataFrames

build running

NOTE: For the latest stable README.md ensure you are on the `master` branch.

Built based on the Apache Arrow columnar memory format, cuDF is a GPU DataFrame library for loading, joining, aggregating, filtering, and otherwise manipulating data.

cuDF provides a pandas-like API that will be familiar to data engineers & data scientists, so they can use it to easily accelerate their workflows without going into the details of CUDA programming.

For example, the following snippet downloads a CSV, then uses the GPU to parse it into rows and columns and run calculations:

```
import cudf, io, requests
from io import StringIO

url="https://github.com/plotly/datasets/raw/master/tips.csv"
content = requests.get(url).content.decode('utf-8')

tips_df = cudf.read_csv(StringIO(content))
tips_df['tip_percentage'] = tips_df['tip']/tips_df['total_bill']*100

# display average tip by dining party size
print(tips_df.groupby('size').tip_percentage.mean())
```

Output:

```
size
```

RAPIDS Release 0.8: Same Community New Freedoms

Making more friends and building more bridges to more ecosystems. It's now easier than ever to get started with RAPIDS.

Josh Patterson
Jul 19 · 7 min read

gQuant—GPU Accelerated examples for Quantitative Analyst Tasks

A simple trading strategy backtest for 5000 stocks using GPUs and getting 20X speedup

Yi Dong
Jul 16 · 6 min read ★

Financial data modeling with RAPIDS.

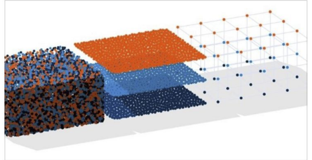
See how RAPIDS was used to place 17th in the Banco Santander Kaggle Competition

Jiwei Liu
Jul 3 · 5 min read

NVIDIA GPUs and Apache Spark, One Step Closer

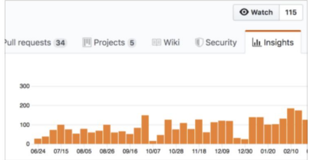
RAPIDS XGBoost4J-Spark Package Now Available

Karthikeyan Rajendran



When Less is More: A brief story about XGBoost feature engineering

A glimpse into how a Data Scientist makes decisions about featuring engineering an XGBoost machine



Nightly News: CI produces latest packages

Release code early and often. Stay current on latest features with our nightly conda and container releases.

https://github.com/rapidsai

https://medium.com/rapids-ai

Getting Started

RAPIDS

How do I get the software?

- https://github.com/rapidsai
- https://anaconda.org/rapidsai/

- https://ngc.nvidia.com/registry/nvidia-rapidsai-rapidsai
- https://hub.docker.com/r/rapidsai/rapidsai/

Join the Movement

Everyone can help!

APACHE ARROW	RAPIDS	Dask	GPU Open Analytics Initiative
https://arrow.apache.org/	https://rapids.ai	https://dask.org	http://gpuopenanalytics.com/
@ApacheArrow	**@RAPIDSAI**	**@Dask_dev**	**@GPUOAI**

Integrations, feedback, documentation support, pull requests, new issues, or code donations welcomed!

THANK YOU

Joshua Patterson
joshuap@nvidia.com
@datametrician

RAPIDS